SEP-18: Startpoints - Manipulating Starting Offsets for Input Streams

Status

Current state: ACCEPTED

Vote: http://mail-archives.apache.org/mod_mbox/samza-dev/201909.mbox/browser

Discussion thread: http://mail-archives.apache.org/mod_mbox/samza-dev/201812.mbox/%3CCABpE9c1_N4hFBsj7Yr5v_7q89SVwnm6Zr-MGuLz1e%3De8MchLkg%40mail.gmail.com%3E

JIRA: ~~SAMZA-1983~~ - SEP-18 : Startpoints implementation in the core framework RESOLVED

***Authors**: Daniel Nimishura, Shanthoosh Venkataraman*

Released: Samza-1.4

Purpose

To provide a common interface for external tools and services to rewind and fast-forward the starting offsets of any input stream of a samza application. This feature will provide the capabilities to manually manipulate the starting offsets by various position types and not only by specific offsets. Many of the current underlying system consumers support different position types for seeking to an offset on an input stream, such as seeks by timestamp, and are not generically exposed by the current framework.

Motivation

A samza application consumes events from multiple input streams, applies transformations and produces the result to a output stream. A samza application is typically comprised of multiple samza containers(physical processes).

If a container of a samza application fails, upon restart it should resume processing events where the failed container had left off. In order to enable this, a samza container periodically checkpoints the current offset for each partition of an input stream. In case of application failures, samza users would want their application to consume from a particular position of a input stream(for instance, either due to a bug in their application or due to a bug in upstream producer of the pipeline). The workflow currently supported in samza to update checkpoints:

1. Users have to manually stop their running samza application.
2. Create a configuration file in XML format and specify the starting offset for each input topic partition.
3. Run the samza-checkpoint tool which updates the checkpoint topic of the samza application with the new user-defined offsets.
4. Users have to manually start their samza application again.

Using the samza-checkpoint tool is tedious and error prone, as it requires proper security access and potentially editing many offsets in a configuration file. In some cases, it can cause a samza container to lose its checkpoints. In addition to the dangerous nature of modifying checkpoints directly, the checkpoint offsets are arbitrary strings with system-specific formats. This requires a different checkpoint tool for each system type (i.e. Kafka, Eventhub, Kinesis, etc...).

Terminology

SSP - System stream partition. For example, on a Kafka stream called SomeEvent in the tracking cluster, the system is tracking, the stream is SomeEvent and the partition is a partition ID in the stream.

JC - Job coordinator.

Requirements

Goals

1. Provide a simple and generic interface to manipulate starting offsets per input stream partition instead of relying on system-specific checkpoint tools and services. This allows flexibility to build REST API layers and tools on top of the common interface.
2. Allow defining starting offsets on an input stream by SSP across all tasks or SSP per task.
3. Framework level support for various offset types such as specific offsets and timestamp-based offsets.
4. Should support the different deployment models of samza viz standalone and yarn. Different API offerings of samza such as beam, SQL, high-level and low-level API should be supported by the solution.
5. Simplicity. Easy for developers and users to create tools and services to set starting offsets of a given Samza job.

Non-goals

1. Rewinding or fast-fowarding state store changelogs in relation to the input streams. The challenge is that the changelog stream is typically log compacted.
2. Providing a service that externally exposes a Startpoint API. Such services require other core changes and will be explored in another SEP or design document.

Proposed Changes

Different systems in Samza have different formats for checkpoint offsets and lack any contract that describes the offset format. To maintain backwards compatibility and to have better operability for setting starting offsets, this solution introduces the concept of Startpoints and utilizes the abstract metadata storage layer.

A Startpoint indicates what offset position a particular SSP should start consuming from. The Startpoint takes higher precedence than Checkpoints and defines the position type and the position value of the position type. For example, if the Startpoint position type is TIMESTAMP, then the position value is an epoch value in milliseconds. The Startpoint enforces a stricter contract for external tools and services to follow as opposed to the string offset value in the Checkpoint.

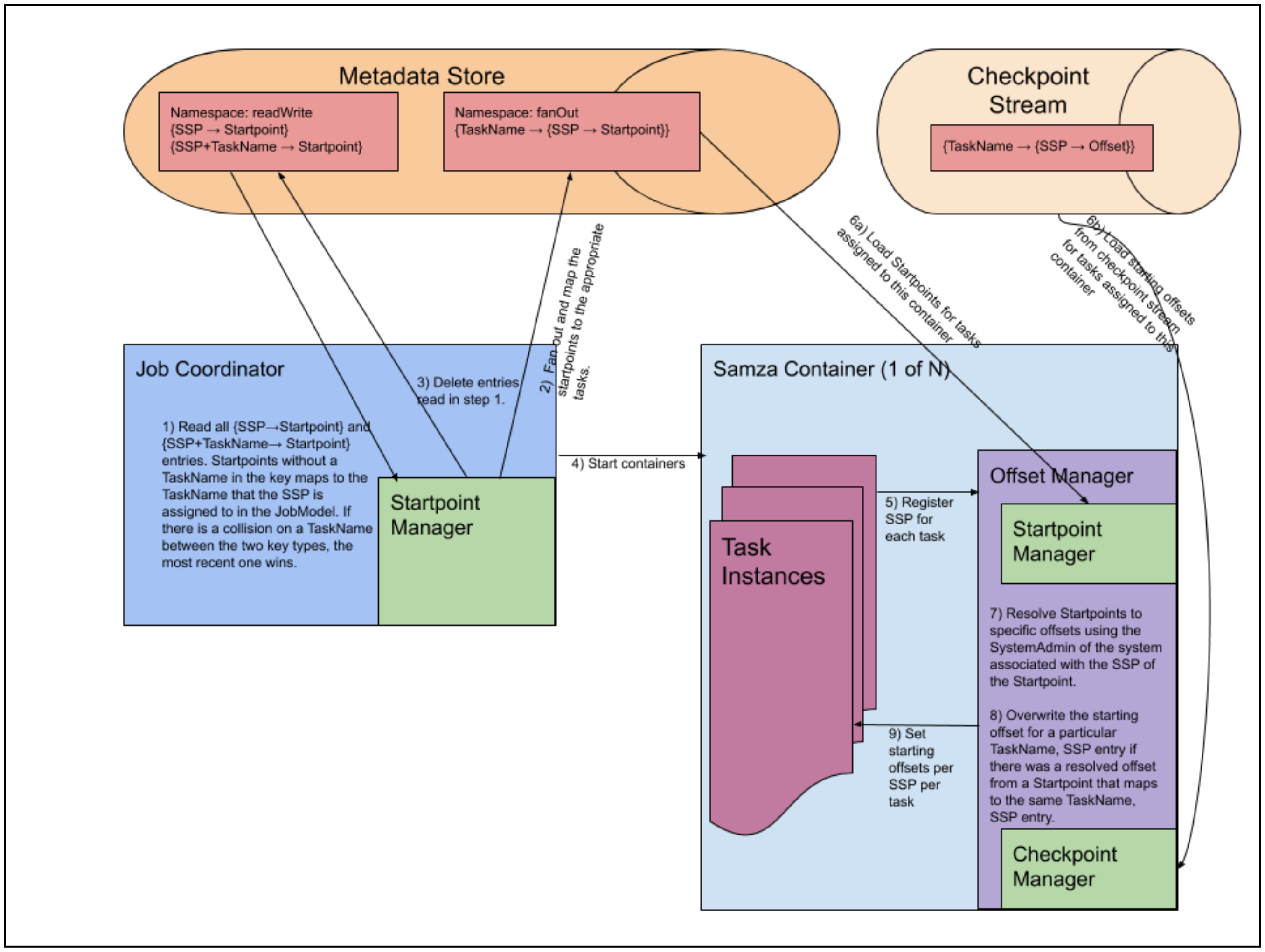
A requested Startpoint will be stored in the metadata store. This will be decoupled from the checkpointed offsets in the checkpoint stream.

General Workflow

Loading Startpoints Upon Job Startup

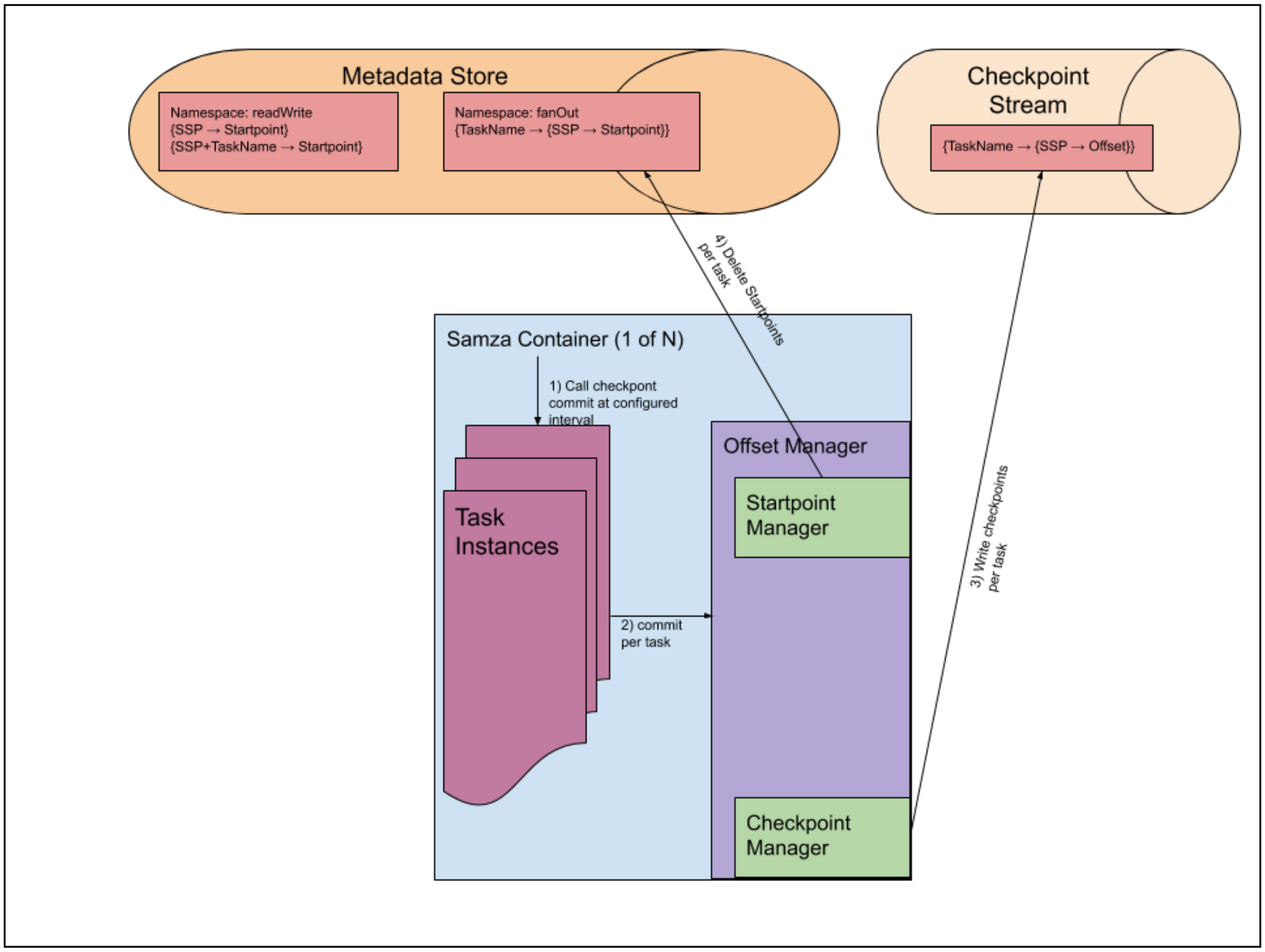
Startpoints are written to the metadata store under a readWrite namespace using two key types: SSP-only and SSP+TaskName. For broadcast input streams, an SSP may span across multiple tasks and therefore, Startpoints are applied at the task level. For Startpoints on SSP-only keys, the task to SSP mappings are retrieved from the JobModel. Upon the subsequent start of the Samza job, the Startpoints are fanned out to a fanOut namespace in the metadata store. The below diagram illustrates the flow.

As with Checkpoints, Startpoints are applied to the starting offset of an SSP in a task instance during the start up time of the SamzaContainer.


Metadata Store
Namespace: readWrite
{SSP → Startpoint}
{SSP+TaskName → Startpoint}
Namespace: fanOut
{TaskName → {SSP → Startpoint}}
Checkpoint Stream
{TaskName → {SSP → Offset}}
Job Coordinator
1) Read all {SSP→Startpoint} and {SSP+TaskName→ Startpoint} entries. Startpoints without a TaskName in the key maps to the TaskName that the SSP is assigned to in the JobModel. If there is a collision on a TaskName between the two key types, the most recent one wins.
2) Fan out and map the startpoints to the appropriate tasks.
3) Delete entries read in step 1.
Startpoint Manager
4) Start containers
Samza Container (1 of N)
Task Instances
5) Register SSP for each task
6a) Load Startpoints for tasks assigned to this container
6b) Load starting offsets from checkpoint stream for tasks assigned to this container
Offset Manager
Startpoint Manager
7) Resolve Startpoints to specific offsets using the SystemAdmin of the system associated with the SSP of the Startpoint.
8) Overwrite the starting offset for a particular TaskName, SSP entry if there was a resolved offset from a Startpoint that maps to the same TaskName, SSP entry.
9) Set starting offsets per SSP per task
Checkpoint Manager

Committing Startpoints

Once a particular Startpoint is applied to the starting offset of a SSP in a task instance, it is subsequently removed at the next offset commit.



Storing Requested Startpoint

Metadata Store

Referenced in the General Workflow above.

The out-of-band metadata store used is described by the metadata store abstraction feature (SAMZA-1786) from SEP-11. The Startpoints are stored within its own namespaces in the metadata store.

Abstractly from the perspective of Startpoint operation, we think of the metadata store as a KV store and mapped as such:

{SSPStartpoint} or *{SSP+TaskNameStartpoint}*

TaskName is optional and is primarily for broadcast inputs where the same SSP spans across multiple tasks.

Upon the fan out, the Startpoints are mapped by task with the same structure as checkpoints:

{TaskName{SSPStartpoint}}

Failure Scenarios

SamzaContainer is restarted before the first offset commit

- The Startpoint will be applied to the starting offset again upon restart. However, this is identical to the same failure scenario for Checkpoints.

JobCoordinator is restarted before all Startpoints are applied

- Similar to the above failure scenario, except across multiple containers since restarting a JobCoordinator restarts all associated SamzaContainers.

Public Interfaces

Startpoint Models

Startpoint

```java
/**
 * Startpoint represents a position in a stream partition.
 */
public abstract class Startpoint {

  private final long creationTimestamp;

  /**
   * Applies the {@link StartpointVisitor}'s visit methods to the {@link Startpoint}
   * and returns the result of that operation.
   * @param input the metadata associated with the startpoint.
   * @param startpointVisitor the visitor of the startpoint.
   * @return the result of applying the visitor on startpoint.
   */
  public abstract <IN, OUT> OUT apply(IN input, StartpointVisitor<IN, OUT> startpointVisitor);
}


/**
 * A {@link Startpoint} that represents the earliest offset in a stream partition.
 */
public final class StartpointOldest extends Startpoint {

  @Override
  public <IN, OUT> OUT apply(IN input, StartpointVisitor<IN, OUT> startpointVisitor) {
    return startpointVisitor.visit(input, this);
  }
}

/**
 * A {@link Startpoint} that represents the latest offset in a stream partition.
 */
public final class StartpointUpcoming extends Startpoint {

  @Override
  public <IN, OUT> OUT apply(IN input, StartpointVisitor<IN, OUT> startpointVisitor) {
    return startpointVisitor.visit(input, this);
  }
}

/**
 * A {@link Startpoint} that represents a timestamp offset in a stream partition.
 */
public final class StartpointTimestamp extends Startpoint {

  private final Long timestampOffset;
}

/**
 * A {@link Startpoint} that represents a specific offset in a stream partition.
 */
public final class StartpointSpecific extends Startpoint {

  private final String specificOffset;

  /**
   * Getter for the specific offset.
   * @return the specific offset.
   */
  public String getSpecificOffset() {
    return specificOffset;
  }

  @Override
  public <IN, OUT> OUT apply(IN input, StartpointVisitor<IN, OUT> startpointVisitor) {
    return startpointVisitor.visit(input, this);
  }
}
```

Additional Interface Method for SystemAdmin

Referred to in Step 7 of the Loading Startpoints Upon Job Startup section above.

SystemAdmin

```
public interface SystemAdmin {
...
  /**
   * Resolves the startpoint to a system specific offset.
   * @param startpoint represents the startpoint.
   * @param systemStreamPartition represents the system stream partition.
   * @return the resolved offset.
   */
  String resolveStartpointToOffset(SystemStreamPartition systemStreamPartition, Startpoint startpoint);
...
}
```

StartpointVisitor

SystemConsumer

```
/**
 * A {@link SystemAdmin} implementation should implement this abstraction to support {@link Startpoint} and the
instance
 * should visit via {@link Startpoint#apply(Object, StartpointVisitor)} within the
 * {@link SystemAdmin#resolveStartpointToOffset(SystemStreamPartition, Startpoint)} implementation.
 */
public interface StartpointVisitor<IN, OUT> {

  /**
   * Performs a sequence of operations using the {@link IN} and {@link StartpointSpecific} and returns the
result of the execution.
   * @param input the input metadata about the startpoint.
   * @param startpointSpecific the {@link Startpoint} that represents the specific offset.
   * @return the result of executing the operations defined by the visit method.
   */
  default OUT visit(IN input, StartpointSpecific startpointSpecific) {
    throw new UnsupportedOperationException("StartpointSpecific is not supported.");
  }

  /**
   * Performs a sequence of operations using the {@link IN} and {@link StartpointTimestamp} and returns the
result of the execution.
   * @param input the input metadata about the startpoint.
   * @param startpointTimestamp the {@link Startpoint} that represents the timestamp.
   * @return the result of executing the operations defined by the visit method.
   */
  default OUT visit(IN input, StartpointTimestamp startpointTimestamp) {
    throw new UnsupportedOperationException("StartpointTimestamp is not supported.");
  }

  /**
   * Performs a sequence of operations using the {@link IN} and {@link StartpointOldest} and returns the result
of the execution.
   * @param input the input metadata about the startpoint.
   * @param startpointOldest the {@link Startpoint} that represents the earliest offset.
   * @return the result of executing the operations defined by the visit method.
   */
  default OUT visit(IN input, StartpointOldest startpointOldest) {
    throw new UnsupportedOperationException("StartpointOldest is not supported.");
  }

  /**
   * Performs a sequence of operations using the {@link IN} and {@link StartpointUpcoming} and returns the
result of the execution.
   * @param input the input metadata about the startpoint.
   * @param startpointUpcoming the {@link Startpoint} that represents the latest offset.
   * @return the result of executing the operations defined by the visit method.
   */
  default OUT visit(IN input, StartpointUpcoming startpointUpcoming) {
    throw new UnsupportedOperationException("StartpointUpcoming is not supported.");
  }
}
```

StartpointManager

StartpointManager is the main API to read and write Startpoints and is composed alongside the CheckpointManager in the OffsetManager. It also provides the methods to fan out the Startpoints. The StartpointManager is system-implementation agnostic and handles the serialization and deserialization of Startpoints into the metadata store.

OffsetManager

```
/**
 * The StartpointManager reads and writes {@link Startpoint} to the provided {@link MetadataStore}
 *
 * The intention for the StartpointManager is to maintain a strong contract between the caller
 * and how Startpoints are stored in the underlying MetadataStore.
 *
 * Startpoints are written in the MetadataStore using keys of two different formats:
 * 1) {@link SystemStreamPartition} only
 * 2) A combination of {@link SystemStreamPartition} and {@link TaskName}
 *
 * Startpoints are then fanned out to a fan out namespace in the MetadataStore by the
 * {@link org.apache.samza.clustermanager.ClusterBasedJobCoordinator} or the standalone
 * {@link org.apache.samza.coordinator.JobCoordinator} upon startup and the
 * {@link org.apache.samza.checkpoint.OffsetManager} gets the fan outs to set the starting offsets per task and
per
 * {@link SystemStreamPartition}. The fan outs are deleted once the offsets are committed to the checkpoint.
 *
 * The read, write and delete methods are intended for external callers.
 * The fan out methods are intended to be used within a job coordinator.
 */
public class StartpointManager {
  /**
   * Writes a {@link Startpoint} that defines the start position for a {@link SystemStreamPartition}.
   * @param ssp The {@link SystemStreamPartition} to map the {@link Startpoint} against.
   * @param startpoint Reference to a Startpoint object.
   */
  public void writeStartpoint(SystemStreamPartition ssp, Startpoint startpoint) {...}

  /**
   * Writes a {@link Startpoint} that defines the start position for a {@link SystemStreamPartition} and {@link
TaskName}.
   * @param ssp The {@link SystemStreamPartition} to map the {@link Startpoint} against.
   * @param taskName The {@link TaskName} to map the {@link Startpoint} against.
   * @param startpoint Reference to a Startpoint object.
   */
  public void writeStartpoint(SystemStreamPartition ssp, TaskName taskName, Startpoint startpoint) {...}

  /**
   * Returns the last {@link Startpoint} that defines the start position for a {@link SystemStreamPartition}.
   * @param ssp The {@link SystemStreamPartition} to fetch the {@link Startpoint} for.
   * @return {@link Optional} of {@link Startpoint} for the {@link SystemStreamPartition}.
   * It is empty if it does not exist or if it is too stale.
   */
  public Optional<Startpoint> readStartpoint(SystemStreamPartition ssp) {...}

  /**
   * Returns the {@link Startpoint} for a {@link SystemStreamPartition} and {@link TaskName}.
   * @param ssp The {@link SystemStreamPartition} to fetch the {@link Startpoint} for.
   * @param taskName The {@link TaskName} to fetch the {@link Startpoint} for.
   * @return {@link Optional} of {@link Startpoint} for the {@link SystemStreamPartition} and {@link TaskName}.
   * It is empty if it does not exist or if it is too stale.
   */
  public Optional<Startpoint> readStartpoint(SystemStreamPartition ssp, TaskName taskName) {...}

  /**
   * Deletes the {@link Startpoint} for a {@link SystemStreamPartition}
   * @param ssp The {@link SystemStreamPartition} to delete the {@link Startpoint} for.
   */
  public void deleteStartpoint(SystemStreamPartition ssp) {...}

  /**
   * Deletes the {@link Startpoint} for a {@link SystemStreamPartition} and {@link TaskName}.
   * @param ssp ssp The {@link SystemStreamPartition} to delete the {@link Startpoint} for.
   * @param taskName ssp The {@link TaskName} to delete the {@link Startpoint} for.
   */
  public void deleteStartpoint(SystemStreamPartition ssp, TaskName taskName) {...}

  /**
   * The Startpoints that are written to with {@link #writeStartpoint(SystemStreamPartition, Startpoint)} and
with
   * {@link #writeStartpoint(SystemStreamPartition, TaskName, Startpoint)} are moved from a "read-write"
namespace
   * to a "fan out" namespace.
   * This method is not atomic or thread-safe. The intent is for the Samza Processor's coordinator to use this
   * method to assign the Startpoints to the appropriate tasks.
   * @param taskToSSPs Determines which {@link TaskName} each {@link SystemStreamPartition} maps to.
   * @return The set of active {@link TaskName}s that were fanned out to.
   */
  public Map<TaskName, Map<SystemStreamPartition, Startpoint>> fanOut(Map<TaskName, Set<SystemStreamPartition>>
taskToSSPs) throws IOException {...}

  /**
   * Read the fanned out {@link Startpoint}s for the given {@link TaskName}
   * @param taskName to read the fan out Startpoints for
   * @return fanned out Startpoints
   */
  public Map<SystemStreamPartition, Startpoint> getFanOutForTask(TaskName taskName) throws IOException {...}

  /**
   * Deletes the fanned out {@link Startpoint#apply(Object, StartpointVisitor)} for the given {@link TaskName}
   * @param taskName to delete the fan out Startpoints for
   */
  public void removeFanOutForTask(TaskName taskName) {...}
}
```

Compatibility, Deprecation, and Migration Plan

All changes are backwards compatible. This is an add on feature and no changes to the existing checkpoint operations are required.

There may be opportunities where offset related APIs that are strongly coupled to Checkpoints may be modified to handle both Startpoints and Checkpoints. Any such APIs will be deprecated until the next major version.

No migration needed for this new feature.

Test Plan

Create a test job that logs the offset of the first event consumed per task and per SSP. Test using all Startpoint types.

Use the test job to test all combinations of:

- Job Coordinators (ex: ClusterbasedJobCoordinator, ZkJobCoordinator, etc..)
- Provided connectors (ex: Kafka, Eventhubs, etc..)
- Broadcast and non-broadcast streams
- Various input streams with various partition counts

Analysis

A key part of the core Startpoint feature is for individual task instances to fetch the appropriate Startpoint keyed by SSP-only. The two approaches, fan-out and intent-ACK, have been explored with the analysis detailed in the following subsections. The fan-out strategy is favored over the intent-ACK strategy. See analysis and explanation below.

Fan-out

See General Workflow above for details

Pros

- Follows the natural progression of the JobCoordinator calculating the job model and then applying the info in the job model to fan out the SSP to SSP-Task Startpoints.
- Cleaner and simpler book keeping of Startpoints. SSP-only keyed Startpoints are deleted after fan out and SSP+TaskName keyed Startpoints are deleted upon offset commits.

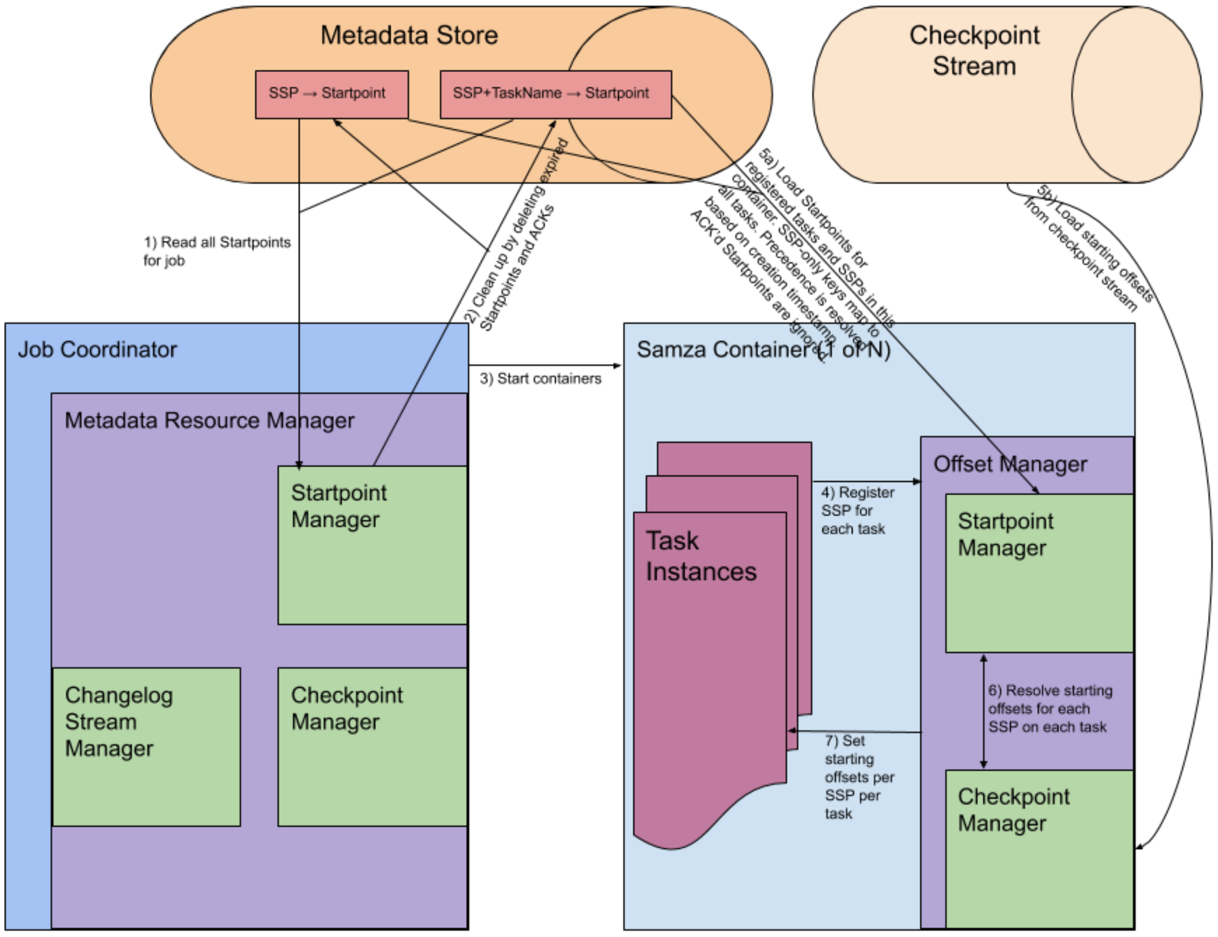
Cons

- Will not work with the PassthroughJobCoordinator because it does not have a leader election strategy. The fan-out approach requires a JC leader. Workarounds will need to be explored during implementation for the small pool of use cases where PassthroughJobCoordinator is needed.

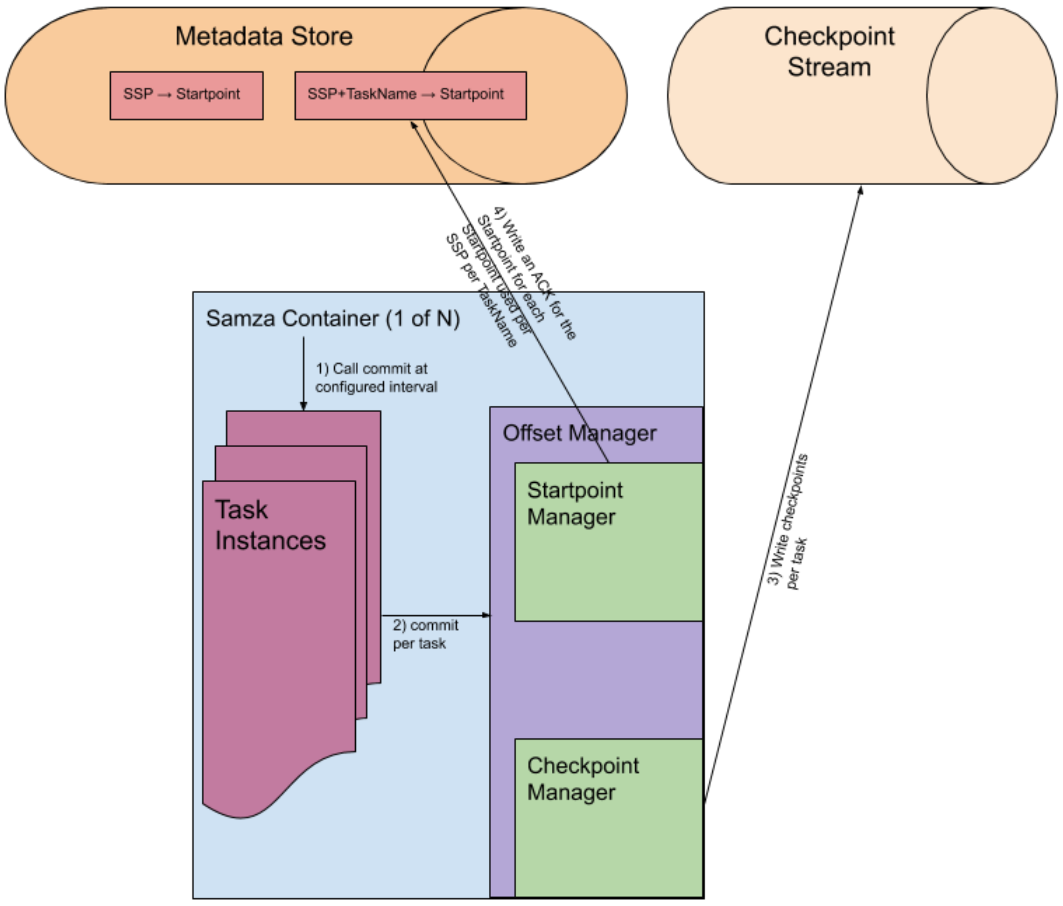
Intent-ACK

This approach is rejected due to the weight of the cons and the complexity required to manage the intents and ACKs once they are applied to the starting offsets and no longer needed.

Loading Startpoints Upon Job Start


Metadata Store
SSP → Startpoint
SSP+TaskName → Startpoint
Checkpoint Stream
1) Read all Startpoints for job
2) Clean up by deleting expired Startpoints and ACKs
3) Start containers
5a) Load Startpoints for registered tasks and SSPs in this container. SSP-only keys map to all tasks. Precedence is resolved based on creation timestamp. ACK'd Startpoints are ignored.
5b) Load starting offsets from checkpoint stream
Job Coordinator
Metadata Resource Manager
Startpoint Manager
Changelog Stream Manager
Checkpoint Manager
Samza Container (1 of N)
Task Instances
4) Register SSP for each task
Offset Manager
Startpoint Manager
6) Resolve starting offsets for each SSP on each task
7) Set starting offsets per SSP per task
Checkpoint Manager

Committing Startpoints


Metadata Store
SSP → Startpoint
SSP+TaskName → Startpoint
Checkpoint Stream
4) Write an ACK for the Startpoint for each Startpoint used per SSP per TaskName
Samza Container (1 of N)
1) Call commit at configured interval
Task Instances
2) commit per task
Offset Manager
Startpoint Manager
Checkpoint Manager
3) Write checkpoints per task

Pros

- Does not rely on Job Coordinator for consumption of Startpoints.

Cons

- Following the consumption of startpoints, metadata store will be polluted with SSP-only keyed startpoints and per-task-ACKs. Current implementation of the metadata store will bootstrap all the startpoints and ACKs until they are cleaned up.
- Task count may change during runtime due to the increase in the input stream partition count. Therefore, new tasks will pick up the Startpoint unless all tasks prior to the increase in task count have written their ACKs and the cleanup process ran prior to the task-to-partition mapping. Startpoints should only apply to the state of the job upon startup and not to any changes during runtime.

Additional Notes

Following is the workflow for the Intent/ACK approach:

1. The startpoint is written to the startpoint store.
2. The SamzaContainer's read the startpoint and acknowledge that they've acted on it.
3. After sufficient number of acks are received, an external agent purge the startpoint. External agent can be either one of the running SamzaContainer of the job or JobCoordinator of the job or a external daemon process.

In above workflow, an external agent will wait for the expected number of ACKs to be published to startpoint-store, and then purge the startpoint. The expected number of ACK's is not a static number.

The container can acknowledge the startpoint at either one of the following two levels of granularity:

1. Container-level: Currently, compared to YARN with static container count(defined by job.container.count), the number of containers in standalone is dynamic. Processors can join and leave the group at any point in time. For standalone, an external agent cannot determine if it had received sufficient number of ACK's with certainty since number of containers is a moving number.

Standalone :

Let's consider the following scenario for standalone application.

A. Four processors are part of standalone application when the external agent watches for number of active processors in standalone application.
B. External agent expects four ACK's to be written to the store before it can purge the startpoint.
C. Before any processor acks the startpoint, due to hardware failure, number of active processors gets reduced to three and a re-balance happens.
D. After re-balance, three live processors acks their corresponding startpoint to the store and are running fine. External agent cannot clear the start-point, since there're only three ACK's when compared to expected four.

Unless we change the coordination protocol in standalone to accommodate start-point, I'm not sure if we can ACK at this level to purge startpoints.

YARN :

There has been interesting recent developments, where we're planning to develop auto-scaling solution for YARN. In the immediate future, the number of containers in YARN will be not fixed and the above scenario described for standalone will apply for YARN as well.

2. Task-level: The number of tasks can change dynamically at run-time either due to increase in partition count of any of the input topics of the job or a new topic has matched the user-defined input topic-regex of the job. The above scenario described for standalone is still applicable here.

Rejected Alternatives

Writing Directly to Checkpoint

Previous explored solutions involved modifying the checkpoint offsets directly. Operationally, the Startpoint solution provides more safety because checkpoints are used for fault-tolerance. To prevent human error, the framework should not allow an external source to manipulate the checkpointed offsets. The ability to set starting offsets out-of-band as Startpoints is designed to do, provides the additional safety layer.

Another pain point for manipulating checkpoints is that it requires the Samza job to be stopped. Startpoints can be written while the job is running.

Startpoint Intent-ACK Model

See Analysis section for details on why the fan-out model is favored.